ARL-TR-7538 ● Nov 2015

US Army Research Laboratory

Enhanced Experience Replay for Deep Reinforcement Learning

by David Doria, Bryan Dawson, and Manuel Vindiola

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-7538 ● Nov 2015

US Army Research Laboratory

Enhanced Experience Replay for Deep Reinforcement Learning

by David Doria, Bryan Dawson, and Manuel Vindiola
Computational and Information Sciences Directorate, ARL

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
November 2015	Final	October 2015

4. TITLE AND SUBTITLE
Enhanced Experience Replay for Deep Reinforcement Learning

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
David Doria, Bryan Dawson, and Manuel Vindiola

5d. PROJECT NUMBER
R.0006163.13

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
US Army Research Laboratory
ATTN: RDRL-CIH-S
Aberdeen Proving Ground, MD 21005-5067

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-7538

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Deep reinforcement learning recently has performed very well in the task of learning control policies for Atari 2600 games. Using raw frames taken directly from an Atari emulator, these systems train a convolutional neural network to interpret the state of the game and select the optimal action. Temporal-difference Q-learning is used to train the network, and a memory of state–action–reward transitions is kept and used in an experience-reply algorithm to increase training efficiency. Recent work reports performance at or above the level of an expert human player in many of the games; however, when evaluating behavior on a more qualitative level, there are major inconsistencies with the actions of an intelligent player. To improve these behavioral characteristics, we introduce 3 new techniques: 1) we bias the experience-replay-selection step toward state transitions that received a positive reward; 2) we compare newly observed states to a set of recently observed states and take a random action rather than accept the action of the current policy if the states are similar to within a threshold; and 3) we only perform the reinforcement learning updates on the topmost linear layers as experiences are generated. This report details these techniques and preliminary results.

15. SUBJECT TERMS
machine learning, reinforcement learning

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Manuel Vindiola
Unclassified	Unclassified	Unclassified	UU	18	19b. TELEPHONE NUMBER (Include area code) 410-278-9151

Contents

List of Figures

List of Tables

1. Introduction

The Army of the future would like to reduce the burden on the Warfighter by deploying autonomous systems on the battlefield. Unfortunately, designing systems that can handle the dynamic and unpredictable conditions on the battlefield is exceedingly difficult. Traditional techniques employ hard-coded rules, definitions, and strategies that cannot adjust to the ever-changing environment. To address these difficulties, researchers have turned to machine learning to model real-world environments and develop systems that can learn and adapt with no prior knowledge, human-encoded assumptions, or additional information about the world. While we may be quite far from a system that can learn and adapt on its own on the battlefield, some researchers have started to study video games as a simpler proxy problem that relies on the same principles. The current state-of-the-art system (Mnih et al. 2015) uses a convolutional neural network to automatically extract relevant features from the video-game display, then uses reinforcement-learning techniques to develop strategies that maximize the game score.

In this report, we provide a detailed description of the existing system, describe several failure modes that we have observed, and present ideas for enhancements that directly address some of the typical failures.

2. Reinforcement Learning

Machine-learning algorithms can be broadly divided into "supervised" and "unsupervised" algorithms. Supervised algorithms rely on large collections of hand-labeled training data and are usually used to solve problems that can be posed as classification or regression tasks. On the other hand, unsupervised learning algorithms are more data-centric—usually trying to automatically find patterns or natural groupings in the data. Reinforcement learning is a particular unsupervised learning method that attempts to develop an optimal action policy by taking actions and observing the consequences of those actions. We will be working with the particular flavor of reinforcement learning called "Q-learning" (Watkins and Dayan 1992). In Q-learning, each state of the system is assigned a Q-value that corresponds to the expected future rewards associated with reaching that state. Initially, the Q-values of all states are set to a random value to encourage exploration of a state's space. The agent chooses its next state, state S`, by choosing the action that leads to the state with the highest associated Q-value. It then updates the Q-value of its current state, S, by adding the immediate reward of choosing state S' with the discounted sum of all possible rewards from being in state S'. This

procedure is described by the Bellman equation, shown in Eq. 1, and is the fundamental equation of reinforcement learning.

$$Q(s,a) = r + \gamma \max_{a'} Q(s',a') \tag{1}$$

When the agent reaches a reward state, the previous state will be updated with real information about the world. Over successive trips through the world toward reward states, the agent transfers that real knowledge to further and further states, developing an optimal action policy along the way.

2.1 Q-Networks

The standard reinforcement-learning procedure maintains an explicit mapping of a particular state to a Q-value that estimates the future reward of being in that state, typically in a table. However, in some cases the state–space is so large that maintaining these explicit mappings is infeasible. In these cases, rather than maintain a table of explicit mappings from states to rewards, an approximation function is used (Baird 1995). One such realization of this function, known as a Q-network, is implemented using a neural network (Tsitsiklis 1997). The network is a function of the state and produces the Q-values of all of the possible actions as output, as shown in Eq. 2. When such a network is updated using the standard back-propagation method (Rumelhart 1986), its weights are adjusted in accordance with the reinforcement-learning policy. For instance, if the optimal action produced by the neural network for a given state generates suboptimal rewards, the weights would be adjusted to diminish the desirability of that action for that state.

$$f(state) = \{action_1, action_2, \ldots, action_i\} \tag{2}$$

2.2 Experience Replay

While the use of a neural network for reinforcement learning is helpful, it does not come without problems. Neural networks are very sensitive to the distribution of the learning data and reinforcement learning can introduce a poor data distribution (Mnih et al. 2015). For example, in an "online" setting, the Q-value of a state is updated in real-time as the agent explores the world. If in a particular region of the state–space there is a dominant optimal action, the network may become biased toward that action as the optimal action even in dissimilar situations. To counteract this problem, a memory of past experiences (the state–action–reward information) is stored during training and the network is updated "offline" using minibatches of randomly sampled experiences. This alleviates the problem of poor data distribution and allows the network to generalize learning across updates. It is

also useful for large state–spaces or in domains where it is difficult to generate new experiences because the agent can repeatedly learn from past experiences.

3. Atari Deep Reinforcement Learning

The current, state-of-the-art, general game-playing system is the deep Q-network (DQN) described by Mnih et al. (2015). It uses a convolutional neural network to interpret frames from Atari 2600 games and define the action policy. It then uses Q-learning and experience replay to update and train the network. The system is connected to an Atari game emulator and the only input to the system is the current frame, the game score, and whether the game is in a terminal state (the player has "lost a life"). At the beginning of every game, the emulator executes a random number of no-operation (NOOP) moves to ensure the game starts in a random configuration. (See Table 1.) After the game information is passed into the system, the emulator returns to the next action to be taken and executes that action—a cycle that is repeated indefinitely until the game terminates. When a game terminates, a new game is started and the observation–action process resumes. In the following subsections we will describe the specific functionality of the Atari deep reinforcement learner in reference to the video game "Breakout".

In this game, the player controls a paddle and uses it to bounce an on-screen ball and break bricks (as shown in Fig. 1a), accumulating a score. The player has 3 lives and only loses a life if the ball "misses" the paddle and contacts the bottom of the screen. If the player loses all 3 lives, the game is over.

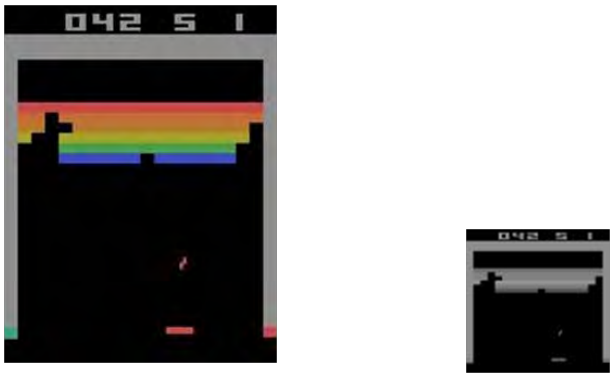
Table 1 Legal actions for the game "Breakout"

Action	Result
NOOP	Do nothing
Fire	Launch the ball
Left	Move the paddle left
Right	Move the paddle right

The learning system has 2 distinct phases to its operation: a random-exploration phase and a learning phase. During the random-exploration phase, the system selects actions at random and simply explores the world. This allows it to build up a random distribution of initial experience to store in its memory for future updates. During the learning phase, the system gradually reduces the number of random actions while gradually increasing the number of decisions made by using the DQN. During this phase, the system also makes periodic updates to the network using minibatches of samples from the experience memory.

3.1 Interpretation

The frame returned from the emulator is 210 × 180 pixels with a 128-color palette (Fig. 1a). To reduce the size of the input space, the frame is passed through a simple preprocessor that scales the frame to 84 × 84 pixels and converts it to grayscale (Fig. 1b). This greatly reduces the dimensionality of the input while preserving most of the necessary information.



a) b)

Fig. 1 Shown a) is a raw frame from "Breakout" and b) a preprocessed frame

The input to the DQN is the state of the game; however, a single preprocessed frame is not sufficient to correctly interpret the game dynamics. For instance, it is impossible to determine the movement of an enemy or the agent from a single preprocessed frame. Therefore, the current state of the game is represented by stacking the previous 3 preprocessed frames with the current preprocessed frame, as shown in Fig. 2. This allows the DQN to extract contextual information from the game and output meaningful action activations. The exact structure and specifications of the DQN can be found in the "Methods" section of Mnih et al. (2005), and a basic diagram of the DQN is shown in Fig. 3.

Fig. 2 Preprocessed frames that compose a state, stacked in temporally descending order

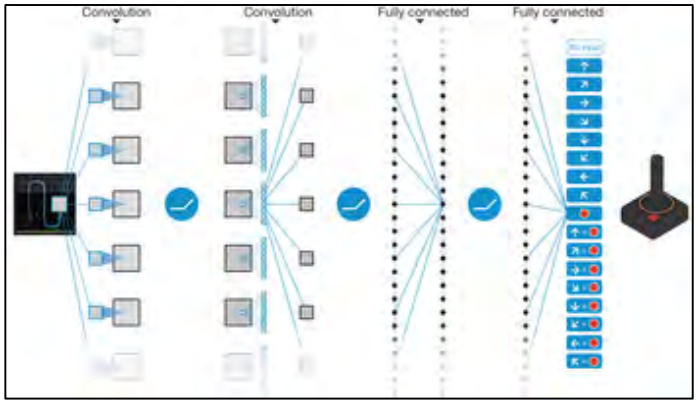


Fig. 3 The basic structure of the DQN

3.2 Experience Storage

An “experience” is defined as a preprocessed frame, the action that frame generated, the reward the action generated, and whether the action generated a terminal state. These values are stored in 4 separate lists and a single experience can be retrieved by accessing the same index across all of the lists. When a minibatch is prepared for a learning update, the system fills a buffer with randomly selected experiences and then takes minibatch-sized chunks from the buffer. The buffer is used until it is empty and then new experiences are randomly selected again. It is important to note the experiences used to fill the buffer consist of stacks of 4 preprocessed frames rather than individual preprocessed frames, as they will be used as inputs to the DQN. In Table 2, we define all of the relevant terms to the learning process.

Table 2 Relevant terms

Frames	$\{frame_1, frame_2, frame_3, ..., frame_i\}$
Actions	$\{action_1, action_2, action_3, ..., action_i\}$
Rewards	$\{reward_1, reward_2, reward_3, ..., reward_i\}$
Terminals	$\{terminal_1, terminal_2, terminal_3, ..., terminal_i\}$
State$_i$	{Frames[i], Frames[i-1], Frames[i-2], Frames[i-3]}
Experience$_i$	{State[i], Rewards[i], Actions[i], Terminals[i]}

3.3 Testing

During the learning phase, we begin updating the DQN and periodically saved the network to a file. To evaluate the performance of the system, the DQN can be loaded from that file and used to play the game (with its weights remaining constant throughout the game play). The system will still save experiences to memory but only so it can generate the 4 frame stacks to pass to the DQN to properly perceive the state of the game. The testing procedure can also display the current frame from the emulator to display the game to the user in real-time, and optionally save the frames to a file for future playback.

4. Contributions

Upon qualitatively evaluating the fully trained system, we noticed a behavior (across several games) that was inconsistent with our goal of automatically learning the objectives and strategies of a game. We found that although the system initially seemed to play the game extremely well, it would eventually get stuck in an infinite loop where the agent would neither die nor continue achieving the game objective (e.g., breaking bricks in the game "Breakout"). This type of behavior indicates that the system has not learned what we would expect it to learn—to play and beat the game—but rather has learned to simply keep playing or to not die. In this section we present several novel modifications to the system that attempt to address this issue.

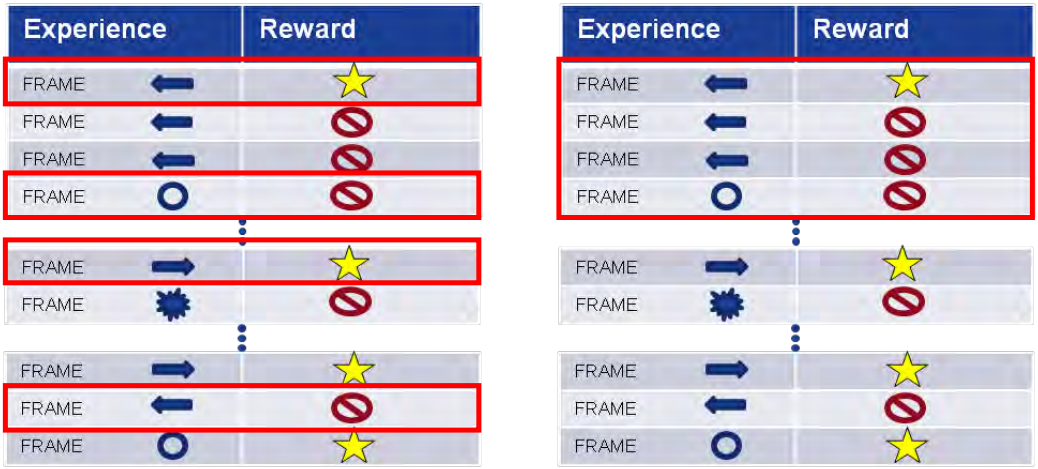
4.1 Reward-Triggered Updates

During the learning process, updates to the network use randomly sampled experiences from the past. In general, there are orders-of-magnitude more training sequences that are unsuccessful and do not lead to rewards than there are successful action sequences. These experiences are all useful to learn general game knowledge (dynamics, etc.), but this does not place emphasis on learning which actions lead to rewards, which is a critical part of learning a strategy. In addition, most games have

a delayed reward signal, meaning there is a temporal gap between the action that actually leads to the reward and when the system experiences that reward. Therefore, when the system learns from an experience that has a positive reward associated with it, the true reward action is actually at some point in the (perhaps recent) past. To reduce the extreme ratio of "successful" to "unsuccessful" training sequences, we introduced reward-triggered updates. This allows the system to intentionally learn directly from the sequence of actions that lead to a reward. When the system experiences a reward, it stores that experience for later use but it also triggers a minibatch update to the DQN using minibatch-sized recent experiences. The update function is defined as $U(N, E)$ where N is a network and E is a set of experiences (see Eq. 3). Then the iterative update equation becomes

$$N(t) = \begin{cases} U(N(t-1), S) & \text{r=0} \\ U(N(t-1), R) & \text{otherwise} \end{cases} \tag{3}$$

where S is the set of experiences in the normal schedule; R is the set of recent experiences; and r is the reward. (Fig. 4 depicts experience selections.)



a) b)

Fig. 4 Pictured in a) is the random experience selection for a scheduled update; pictured in b) is the selection of recent experiences for a reward-triggered update.

4.2 Repetition Inhibition

As noted earlier, we found the system tended to get stuck in infinite action loops during testing. However, since each iteration of training is equivalent to a testing procedure, the system experiences these loops during training as well. While there is a small chance of taking a random action in later stages of training, if the system were to get stuck in an infinite loop, it would likely persist for long enough that the

experience memory would be saturated with the same subset of experiences. In turn, the DQN would be updated using that same subset of experiences and would cease learning anything useful until the system found its way out of the loop. To address this, we check the current preprocessed frame against recently encountered, preprocessed frames. If the distance (Eq. 4: L2 pixel-wise distance) between the current frame and a previous frame is below a threshold, rather than follow the current policy we would take a random action and attempt to break out of the loop. This choice is described in Eq. 5 and illustrated in Fig. 5.

$$D(F1, F2) = \sum_{r=1}^{R} \sum_{c=1}^{C} F1(r, c) F2(r, c) \tag{4}$$

$$action(s) = \begin{cases} random() & \min_{i \in R} D(s, frame_i) < \epsilon \\ policy(s) & \text{otherwise} \end{cases} \tag{5}$$

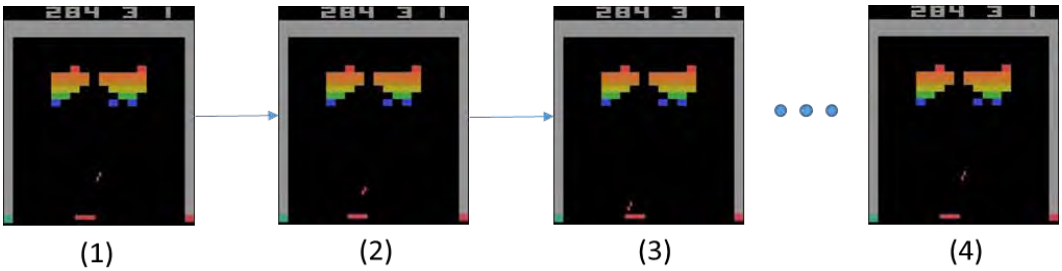


Fig. 5 A sequence in which Frames 1 and 4 are identical, indicating a loop; rather than follow the action policy for Frame 4, a random action would be taken

4.3 Representation Pretraining

The convolutional neural network employed by the system is responsible for 2 distinct operations: interpreting the visual output of the game (convolutional layers) and defining the action policy (linear layers). When the network is updated using standard error back-propagation, the weights throughout the entire network are adjusted. This update procedure therefore is simultaneously trying to learn a good representation and a good action policy. During the experience-replay phase, the system learns with past experiences as inputs on the assumption the network has updated its learning policy (but not its representations); so, there may be more information to be gathered by examining those past experiences again. However, since the representations have also been changed, there is now a difference in how the system perceives those past experiences, reducing the amount that can be learned from the updated learning policy.

To address this, we train a full network until it reaches an acceptable level of performance and then separate the convolutional layers from the remainder of the network. We then reinitialize and train the linear layers using the separated convolutional layers as a fixed subnetwork whose output is used as the input to the linear layers responsible for defining the action policy, as shown in Fig. 6. When an update occurs, we only adjust the linear network, which modifies the action policy without the requirement of relearning the representations. Once the representations have been learned once, this technique results in a substantial decrease in training time, from 12 days to only 1 day, allowing further experimentation without enormous delays in the design and testing cycle.

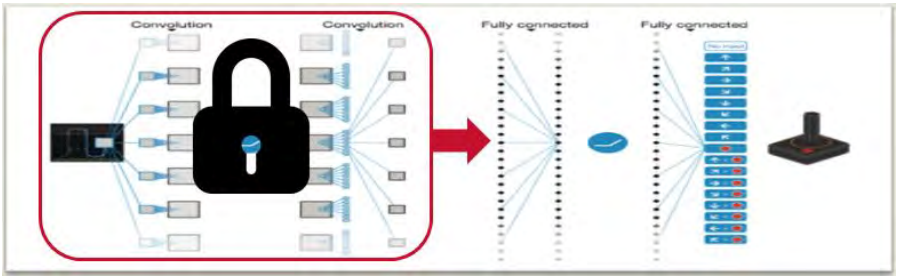


Fig. 6 The fixed subnetwork of convolutional layers feeding into the reinitialized sub-network of linear layers

5. Future Work

In future work, we will develop and perform quantitative-analysis techniques to better understand the effects of these modifications on the system. For example, we will examine the reward distributions (the number of times the system reaches some number of rewards before terminating) to determine which regions of the state–space the game is in during the majority of the training. We will also evaluate these techniques on more games, where we would expect to see a greater increase in performance as the complexity of the games increase. Finally, we will explore additional ideas that specifically address the problem of the delayed reward signal and how to correlate sequences of actions with rewards.

6. Conclusions

We have detailed the current state-of-the-art system for automatically learning action policies to play simple video games. After noting a serious shortcoming of the current system, we have introduced and studied 3 enhancements to mitigate this problem and have given a preliminary evaluation of their effect. Though we have not yet observed dramatic improvements in performance, we are hopeful that these techniques—coupled with future improvements—will enable the system to obtain a higher level of awareness and knowledge about the game dynamics and goals.

7. References

Baird L. Residual algorithms: reinforcement learning with function approximation. Proceedings of the 12th International Conference on Machine Learning; 1995; San Francisco (CA). p. 30–37.

Mnih V, Kavukcuoglu K, Silver D, Rusu AA, Veness J, Bellemare MG, Graves A, Riedmiller M, Fidjeland AK, Ostrovski G, et al. Human-level control through deep reinforcement learning. Nature. 2015;518:529–533.

Rumelhart DE, Hinton GE, Williams RJ. Learning representations by back-propagating errors. Nature. 1986;323:533–536.

Tsitsiklis JN, Van Roy B. An analysis of temporal-difference learning with function approximation. IEEE Transactions on Automatic Control. 1997;42(5):674–690.

Watkins CJCH, Dayan P. Q-learning. Mach Learn. 1992;8:279–292.

1 (PDF) DEFENSE TECHNICAL INFORMATION CTR DTIC OCA

2 (PDF) DIRECTOR US ARMY RESEARCH LAB RDRL CIO LL IMAL HRA MAIL & RECORDS MGMT

1 (PDF) GOVT PRINTG OFC A MALHOTRA

2 (PDF) DIR USARL RDRL CIH S J INFANTOLINO M VINDIOLA

INTENTIONALLY LEFT BLANK.